UNCLASSIFIED

AD NUMBER

ADB206370

NEW LIMITATION CHANGE

TO

Approved for public release, distribution unlimited

FROM

Distribution authorized to U.S. Gov't. agencies and their contractors; Critical Technology; Nov 95. Other requests shall be referred to AFMC/STI, Phililips Lab., Kirtland AFB, NM 87117-5776.

AUTHORITY

Phillips Lab [AFMC], Kirtland AFB, NM ltr dtd 28 Jul 97

THIS PAGE IS UNCLASSIFIED

PL-TR--95-1093 PL-TR--95-1093

REUSABLE REAL-TIME OPERATING SYSTEM FOR SPACECRAFT OPERATING SYSTEM SPECIFICATION

LORAL Federal Systems -- Manassas
9500 Godwin Drive
Manassas, VA 22110

November 1995

Final Report

Distribution authorized to U.S. Government Agencies and their contractors only; Critical Technology; November 1995. Other requests for this document shall be referred to AFMC/STI.

WARNING - This document contains technical data whose export is restricted by the Arms Export Control Act (Title 22, U.S.C., Sec 2751 et seq.) or The Export Administration Act of 1979, as amended (Title 50, U.S.C., App. 2401, et seq.). Violations of these export laws are subject to severe criminal penalties. Disseminate IAW the provisions of DoD Directive 5230.25 and AFI 61-204.

DESTRUCTION NOTICE - For classified documents, follow the procedures in DoD 5200.22-M, Industrial Security Manual, Section II-19 or DoD 5200.1-R, Information Security Program Regulation, Chapter IX. For unclassified, limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.

PHILLIPS LABORATORY
Space and Missiles Technology Directorate
AIR FORCE MATERIEL COMMAND
KIRTLAND AIR FORCE BASE, NM 87117-5776

PL-TR--95-1093

This final report was prepared by Loral Federal Systems -- Manassa, VA under contract F29601-93-C-0184, Job Order 3672TBAK. The Laboratory Project Officer-in-Charge was Capt Mary Boom (VTQ).

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

If your address has changed, if you wish to be removed from the mailing list, or if your organization no longer employs the addressee, please notify PL/VTQ, Kirtland AFB, NM 87117-5776, to help maintain a current mailing list.

This technical report has been reviewed and is approved for publication.

MARY M. BOOM, Capt, USAF
Project Officer

FOR THE COMMANDER

CHRISTINE M. ANDERSON, GM-15
Chief, Satellite Control & Simulation Division

HENRY L. PUGH, JR., Col, USAF
Director of Space and Missiles Technology

DO NOT RETURN COPIES OF THIS REPORT UNLESS CONTRACTUAL OBLIGATIONS OR NOTICE ON A SPECIFIC DOCUMENT REQUIRES THAT IT BE RETURNED.

DRAFT SF 298

1. Report Date (dd-mm-yy)
November 1995

2. Report Type
Final

3. Dates covered (from... to)
04/94 to 7/95

4. Title & subtitle
Reusable Real-Time Operating System for Spacecraft
Operating System Specification

5a. Contract or Grant #
F29601-93-C-0184

5b. Program Element # 63756D

6. Author(s)

5c. Project # 3672

5d. Task # TB

5e. Work Unit # AK

7. Performing Organization Name & Address
Loral Federal Systems -- Manassas
9500 Godwin Drive
Manassas, VA 22110

8. Performing Organization Report #

9. Sponsoring/Monitoring Agency Name & Address
Phillips Laboratory
3550 Aberdeen SE
Kirtland AFB, NM 87117-5776

10. Monitor Acronym

11. Monitor Report #
PL-TR--95-1093

12. Distribution/Availability Statement Distribution limited to U.S. Government agencies and their contractors only; Critical technology; November 1995. Other requests for this document shall be referred to AFMC/STI.

13. Supplementary Notes

14. Abstract This document is the software requirements specification (SRS) for a Reusable Ada-based Real-time Operating System (RROSS) sponsored by Phillips Laboratory at Kirtland Air Force Base and the Ada Joint Program Office. This specification is part of a study by USAF Phillips Laboratory to create an open standard for a spacecraft operating system and write standard interfaces to the Ada run-time system. The goal of this study is to determine what OS features, capabilities and standard interfaces to payload and bus software are required or desired by spacecraft software developers and to create a specification based on these needs. This SRS specifies an operating system for a processor embedded within a spacecraft. Computer or program failures can mean loss of mission goals for even overall mission failure.

15. Subject Terms Realtime, Operating system, Specification

Security Classification of

16. Report
Unclassified

17. Abstract
Unclassified

18. This Page
Unclassified

19. Limitation of Abstract
Limited

20. # of Pages
44

21. Responsible Person (Name and Telephone #)
Capt Mary M. Boom
(505) 846-0461 Ext 317

CONTENTS

FIGURES

1.0) INTRODUCTION

1.1) Purpose

This document is the software requirements specification (SRS) for a Reusable Real-time Operating System (RROSS) sponsored by Phillips Laboratory at Kirtland Air Force Base and the Ada Joint Program Office. This specification is part of a study funded by Phillips Laboratory to create an open standard for a spacecraft operating system. The goal of this study is to determine what OS features and capabilities are required or desired by spacecraft software developers and to create a specification based on these needs.

Requirements are taken directly or derived from the Statement Of Work (SOW), technical interchanges with spacecraft developers and operating system vendors, and from experience with the Advanced Spaceborne Computer Module (ASCM) program. The objective of the ASCM program, sponsored by Phillips Laboratory, is to develop and space qualify two 32-bit processor modules for data and signal processing applications in future space systems to dramatically increase the satellite onboard processing capability. Included in the development are space qualified higher throughput 32-bit processors, higher density memories, advanced packaging to decrease the overall size of the electronics packages, and supporting software. ASCM provided operating system development experience for the Control Processor Module (CPM) phase and the Advanced Technology Insertion Module (ATIM) phase has already generated a user's requirements data base compiled and maintained at LORAL Federal Systems - Manassas.. This SRS is designed to be an aid to the user in choosing an operating system as well as a constraint on the operating system developer.

1.2) Scope

An operating system for processors embedded within spacecraft will be specified. The operating system is to be portable in two ways: users must be able to reuse programs on multiple platforms without changing the calls to system routines, and the operating system itself must be easily reused across multiple architectures. Features will be identified as *essential*, required to meet specification; or *optional*, desired but exceeds the specification. The origin of requirements will also be identified in order to provide context.

1.3) Definitions, Acronyms, and Abbreviations

ANSI	American National Standards Institute
AP	Application Program
API	Application Program Interface
ASCM	Advanced Spaceborne Computer Module
ATIM	Advanced Technology Insertion Module
CDRL	Contract Data Requirements List; refers to documents required by a DOD contract to be delivered to the government.

consistent	In the context of data, consistent refers to the trait of being valid and up to date. If one part of a structure has been updated and another part is affected, but not yet updated, the structure is said to be *inconsistent*.
CPM	Control Processor Module
CPU	Central Processing Unit
DMA	Direct Memory Access; refers to a device other than the CPU accessing memory without CPU intervention.
DOD	United States Department of Defense
FTP	File Transfer Protocol; standard TCP/IP protocol
host	The workstation or computer which acts as a software development station, i.e. runs compilers and linkers and provides a debug interface to the target processor.
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol; refers to the Internet addressing and packet definitions which are built upon hardware protocols and then form the basis for all Internet communications programs. This is the interface between the transport layer and physical layer.
KB	Kilobytes
MB	Megabytes
MIPS	Million Instructions Per Second
ms	millisecond, 1/1000 second
NFS	Network File System; refers to programs which allow named storage groups (files) to be accessed from network connected devices.
NVRAM	Non-Volatile Random Access Memory; refers to RAM which holds a value without being powered.
OS	Operating System
POSIX	Portable Operating System Interface (IEEE-Std-1003)
RAM	Random Access Memory
Real-time	When a system's response to a stimulus is valid for only a short time (typically seconds or less) due to further changes in environment and stimuli it is said to be a real-time system.
reliable	In the context of network communications protocols, reliable refers to transfers for which the sender gets acknowledgment and the receiver checks the order and number of the transmitted packets.
ROM	Read Only Memory
RPC	Remote Procedure Calls; refers to TCP/IP programming interface to execute subprograms on a remote host.

RROSS	Reusable Real-time Operating System for Spacecraft
shall	Refers to a feature which is required in order to meet the minimum criteria of this specification; legally binding.
should	Refers to a feature which is desired but not required to meet the criteria of this specification.
SLIP	Serial Line Internet Protocol; allows two TCP/IP hosts to connect between serial ports.
SOW	Statement Of Work; refers to the portion of a contract which clearly states the expected results to be provided by the vendor to the buyer.
SRS	Software Requirements Specification
target	Embedded computer which will execute RROSS and user programs developed for RROSS.
TCP	Transmission Control Protocol; refers to a *reliable* network transfer protocol in which user programs may assume the consistency of the data.
TCP/IP	Transmission Control Protocol/Internet Protocol; refers specifically to TCP layered onto IP [See TCP], and generally to standard Internet communications.
Timely	Refers to a response which occurs soon enough after a stimulus such that the response is still valid for the current environment; refer to Real-time.
TOD	Time Of Day; refers to free running processor clock which is used to track the actual time elapsed since it was set.
UDP	User Datagram Protocol; refers to an *unreliable* network transfer protocol in which user programs are responsible for the consistency of the data.
UNIX	Multitasking operating system supporting multiple users originally developed at AT&T Bell Laboratory.
unreliable	In the context of network communications protocols, unreliable refers to transfers for which the sender gets no acknowledgment so that it is unknown whether all or any of the transmitted packets arrived safely.

1.4) References

IEEE-Std-830-1993: IEEE Recommended Practice for Software Requirements Specification

IEEE-Std-1003.1-1990: POSIX Part 1: System Application Program Interface (API) - C Language

IEEE-Std-1003.4-1993: POSIX Part 1: API - C Language - Amendment: Real-time Extensions

IEEE-Std-1003.5-1992: POSIX Ada Language Interfaces - Part 1: Binding for System Application Program Interface

ANSI/ISO/IEC-8652:95: Information Technology - Programming Languages--Ada: Ada Reference Manual

MIL-Std-1815A-1983: Ada Programming Language

Statement Of Work - Contract Number F29601-93-C-0184: Statement of Work for Development of this specification for United States Air Force, Phillips Laboratory, Kirtland Air Force Base, New Mexico.

1.5) Overview

The requirements in this document are based upon the contract's Statement Of Work, technical interchanges with spacecraft developers and operating system vendors, and the work done for the ASCM project. The format of this specification follows ANSI and IEEE Std 830-1993 starting with an introduction, then a high level description of the specified product, followed by actual requirements organized by function. The introduction provides the reader with a brief description of the product, the part it plays in the system, and references used. The high level description provides the reader with a more detailed description of the product without presenting any actual requirements. The purpose of the high level description is to provide background information and a context in which to apply the binding requirements specified in the third section. Specific requirements in section three may be essential or secondary. Essential requirements are specified with *shall*; secondary requirements are specified with *should.* Secondary requirements are those which are desirable, but are not needed to fulfill spacecraft requirements.

2.0) OVERALL DESCRIPTION

2.1) Product Perspective

This Software Requirements Specification (SRS) specifies an operating system for a processor embedded within a spacecraft. Programs executing within a spacecraft control the thrusters, gyroscopes, science instruments, directional sensors, and communication devices. Large amounts of data are often stored, processed, and transmitted. Computer or program failures can mean loss of mission goals or even overall mission failure. The specified operating system, RROSS, must provide the portability, reusability, and open standards available to users of commercial operating systems while providing, or at least allowing, the data protection, error detection and recovery, and reliability of military and spaceborne software systems.

2.1.1) System Interfaces

An operating system is supported by development tools ranging from a compiler and linker to an integrated suite including edit, compile, link, profile, simulate, and debug capabilities. The operating system may come with a customized set of tools or rely on other commercially available tools with the addition of customized libraries or link addresses. This specification will specify the support which is necessary for developing user programs for an embedded target processor.

2.1.1.1) Development Environment

A user must be able to create or update source files with any host system editor, even if one is provided with the development system. The development system must support the compilation of Ada, C, and assembler source code. Users must be able to link the object modules together and create an executable made up of subprograms written in different languages. Error messages and warnings must be provided, including warnings when the user has replaced a function provided by the system or standard library with a user function. The linker must provide the ability to specify the location of user programs when the operating system (or load support tool) loads them into memory.

Configurability of system software should also allow users to specify which pieces of the operating system will be loaded and used in the system. For example, one should be able to choose network support for specific devices (e.g. RS232 interface) and not have to choose all network support or none.

Optionally, the development tools may include a profiler to analyze execution, performance, and data flow within a program, and a simulator to execute programs on the development system as if it were on the target processor. The simulator should optionally support source level and symbolic debugging as well as instruction level execution and visibility. (Reference 3.1.1)

2.1.1.2) Debug Environment

Users must have an interactive path from the development workstation to the operating system running on the target processor. It must support loading, executing, inserting faults, and monitoring the operating system and user programs.

The debug tools must support loading the target processor with executable files produced by the linker tools. The user should not have to manipulate the executable files to specify addresses, sizes, or other parameters required to place the program.

Low level debug interfaces will not be specified since they are part of the hardware platform on which the operating system executes. The operating system and the user programs are equally dependent upon a good instruction level debug path in the processor. Not all processors support a non-intrusive debug port, but it is recommended. Such a port should allow programs to be run, stopped, and stepped by an instruction; memory and registers to be accessed, interrupts to be set, and breakpoints to be used. Disassembly of instructions from memory should also be supported.

The operating system should support high level debug techniques. In particular, a user should be allowed to run a program and retrieve the contents of variables and memory locations, information on which routines have been called, and the contents of parameters given (in) and returned (out). Interprocess communications and relationships should also be visible such as contents of message queues, current status of all active tasks, why tasks are blocked, and status of controlled resources. Multi-processor operating systems should allow the examination of interprocessor communications structures and status of tasks running on other processors. It is especially important to support strong debug tools when using a processor implementation which does not support non-intrusive debug at the instruction level.

The development system should also support *patching* of programs (writing object code directly into memory or binary images to be loaded). This practice is not uncommon (in the context of *fixing* a program in a computer millions of miles from Earth using a relatively slow load path) to minimize upload time and impact to the rest of the memory image. The user will need to see object code produced from the compiles along with detailed maps of the placement of each routine by the linker. Finally, the operating system must allow program images to be updated while the system is running since (unless the hardware supports an independent memory access path) a communications program will receive the command to update memory and then the data and location to be changed. This is described in section 2.1.7.3 of Operations. The ability to update data, programs, and system structures may also be used to insert errors into the software in order to test the system's response and recovery. (Reference 3.1.2)

2.1.2) User Interfaces

Users interact with the operating system through the system interfaces such as the debug and development environments. The system interfaces should provide user friendliness with mouse directed point-and-click menus. All actions provided to the user should be available on a menu along with on-line help information. When developing software, the developer should be able to enter code with a favorite editor, build the code within the development environment, have compile and link errors highlighted and available for update, rebuild until successful, simulate the program, download the program to the operating system running on the target, and run the program. Preferably, except for writing code, only the mouse is used to perform these actions. The development environment is dependent more upon the compiler and tool suppliers than the operating system suppliers. It is a major factor when considering productivity and quality of user developed software. (Reference 3.1.1)

2.1.3) Hardware Interfaces

Since no particular hardware architecture may be specified by this document, the requirements in this section are valid only for architectures which support the hardware described.

The operating system must handle or allow the user to handle interrupts caused by parity, timing, or other types of errors. The operating system must allow response to any and all error interrupts provided by the hardware. (Reference 3.2.1.9)

The operating system must allow CPU reset, when presented as an interrupt, to be handled by the system initialization program whether or not it is part of the operating system. (Reference 3.5.2.2)

The operating system must allow for the implementation of a *deadman* timer switch which will behave as a CPU reset when it expires. Resetting of the timer must be supported and must be customizable by the user to handle any hardware implementation.

Background health and status processing must be allowed. The operating system or user programs must be allowed to *scrub* memory periodically where memory error correction is supported. The operating system or user programs must be allowed to perform monitoring functions to determine whether all tasks are running normally. (Reference 3.5.3.2)

Where hardware support exists for replacing failed memory devices with spare devices (such as column sparing), the operating system should provide interfaces or services to detect failures and replace the device. Where interfaces or services are not provided, the operating system must not prohibit user programs from receiving memory error reports and performing the replacement. (Reference 3.2.1.2)

2.1.4) Software Interfaces

The software interfaces are the Application Program Interfaces. They provide standard procedure and function calls to user programs to request operating system services such as message passing, device access, or timing. May include device driver interfaces and redundancy for data protection.

The interface to the operating system from software will be by way of system library routines or language interfaces to system procedures and functions. Inter-language interfaces desired include Ada to C, C to Ada, Ada to assembler, and C to assembler. Each interface provided must define the register, stack, or memory preparation preceding the call to a function written in another language. Rules for memory usage must also be provided so users can prevent memory allocation problems between Ada Runtime Executives and C and assembler programs. Because the tasking properties of Ada-95 are nondeterministic during rendezvous, Ada programs must be allowed to run with a multitasking kernel or the implemented Ada Runtime Executive must be deterministic. In either case, the rules governing rendezvous or dispatching must be documented for users. (Reference 3.4.1.2)

2.1.4.1) Software Reusability

It is required that the operating system offer reusability on two levels:

- The operating system must be portable to many processors.

- The applications must be able to run with this operating system on any processor by merely recompiling with the appropriate compiler linker tools. The software interfaces must be the same no matter what processor is acting as host to the software.

(Reference 3.5.5)

2.1.4.2) Software Standards

The operating system is expected to conform as closely as possible to standard Ada (MIL-Std-1815A-1983) or POSIX (IEEE-Std-1003.1) with a goal and documented migration path toward the upcoming Ada-95 (ANSI/ISO/IEC-8652:95) and POSIX Ada Bindings (IEEE-Std-1003.5), or POSIX Real-time Extensions (IEEE-Std-1003.4). (Reference 3.4.1.1)

2.1.4.3) Software Capabilities

Capabilities may be provided to the user in a variety of forms. Many math functions, string functions, bit operators, and other low level data manipulations are typically provided as subroutines in a library. Other capabilities, such as sending and receiving messages, device I/O, or controlling dispatching, may be provided as interfaces to kernel routines. Recommendations will be made in some cases with regard to partitioning of function between library subroutines and system routines, but this specification will not attempt to design the product by making requirements of this nature.

Math function libraries, string manipulation libraries, bitwise operations libraries, dynamic memory allocation and deallocation functions, and interprocess communications functions are to be provided. (Reference 3.4.1.2)

Parameter checking on may be done by the function being called, but it is not required and may be done by compiler generated code.

2.1.4.4) Software Customization

User programs must be allowed to change, replace, or create new and existing capabilities to meet specific needs. Outside of the core kernel services, functions will be specified as expandable, meaning that a user may replace functions or add new ones which can be used as system calls or library routines by other user programs. (Reference 3.5.6)

2.1.5) Communications Interfaces

Communications interfaces will be specified at the device driver level. Since no range of system architectures has been specified, the types of communications is not known. This specification concentrates on the definition of user interfaces for communications protocols such as setup, open, read, write, and close. (Reference 3.2.1.4)

Communication interfaces and device drivers must at least report detected errors. The errors may be with protocol, timing, or parity. Opening a communications port should establish that an error-free connection exists. Closing a port must clean up any data structures used by the port even when following an error. (Reference 3.2.1.5)

Other support should include reporting to any additional processes which have registered for errors on a specific device; logging errors for later retrieval; buffering data until it has been assured of reaching its destination; or retransmitting unacknowledged data. Error recovery support may include information queries such as reporting the current owner or user of a device or all devices owned by a process.

To support processes when hardware devices fail, the device driver should be able to detach from a failed device, attach to another device, and provide service under the same device name as before.

To support redundant processes, a device should support sending received data to multiple processes simultaneously. Each process may then interpret the data independently and check on another's results. The reverse is also desired which is to allow multiple processes to send data simultaneously with only matching data being sent out and mismatched data causing an error status to be returned to the callers.

To support redundant data paths, a device driver should be allowed to send and receive data via multiple hardware devices while providing the user program with a standard interface. When writing or sending, the device driver would transmit the same information on all of the attached hardware ports so that the receiver could be assured of receiving at least one good copy. When reading or receiving, the device driver would receive data from all of the attached hardware ports; compare the copies, and; if the data met the correct criteria, pass the data to the user. Data which did not compare should be reported as an error condition. Such a driver would allow compare or voting algorithms to be used on communications without each user program needing to implement them. (Reference 3.2.2.2)

2.1.6) Memory Constraints

Due to the expense of radiation hardened memory components, spacecraft have small RAM spaces usually between 64KB and 64MB. ROM space is even more scarce, 64KB being the usual maximum. Memory must be efficiently allocated and deallocated and the operating system and user programs must be small enough to fit within the target system. (Reference 3.4.2)

2.1.7) Operations

A spacecraft will have several user programs operating simultaneously to control the spacecraft and its instrumentation. Sensing, communication, power control, navigation, and pointing programs will all need to run within strict time limits and have access to communication connections to other spacecraft systems. User programs expect to operate in a way that allows them to share data easily and quickly. They must be guaranteed timely access to critical resources, device or data.

2.1.7.1) Initialization

The operating system should have as short an initialization time as possible. Any state data, hardware or software, that is assumed to be initialized at operating system startup must be reinitialized for restarting the system without reloading all of the software. Programs must not be prohibited from saving state data to be retrieved later, even after a restart of the operating system, in order to return to a predefined state. The operating system must allow forward checkpointing under the direction of user programs. (Reference 3.5.2.2)

The operating system or user programs should be allowed execute from ROM or RAM.

2.1.7.2) Program Execution

Most spacecraft operate with strict timelines for events to occur. The operating system must be able to guarantee completion of a cyclic task with periods as short as 8ms or as long as a day. Aperiodic functions, such as interrupts, or communications processing must be started within times as short as 1ms after receiving the stimulus. The actual time to complete the task will depend on the user programs.

Interprocess communication must support both large and small data packets to allow information sharing and coordination of tasks. Data sharing protected with atomic operations must also be supported. (Reference 3.2.1.1)

2.1.7.3) Code Updates

Debugging problems after launch require visibility into the system by communications programs. Just as usual debug on the ground requires visibility into memory, variables, program status, resource availability, and message queues, so does debug of the deployed system. Since the only available communication path for a deployed embedded processor is through communication paths, programs monitoring those paths must be able to access the required debug information. Access may be provided by system subroutines, shell tasks, or other query mechanism. The user should be allowed to create a customized query routine built from system level subroutines or functions.

A user program, receiving commands from the ground via a communications link, must be able to update program images in RAM. Programs will either be updated after loading or updated and reloaded. A user program controlled from the ground will also be able to read or write memory in order to support remote debugging via a communication channel. Allowing visibility and control from the ground is a major feature. (Reference 3.2.2.3)

To protect data during flight, RROSS should provide memory protection. When the code is being updated, memory protection would have to be disabled for the blocks being changed. Once the update is complete, memory protection should be enabled again on all code blocks. (Reference 3.2.1.2)

2.1.8) Site Adaptation Requirements

The system must allow downloading to a processor embedded within a spacecraft subsystem. The target processor will not have a display or keyboard. The user will use signal probes, discrete input and output signals, and a CPU test interface such as IEEE-1149.1 or IEEE-488. It is desirable that the development system and operating system support function traces, maps of procedure and variable locations, and debugging through an I/O device. (Reference 3.4.2)

2.2) Product Functions

2.2.1) Real-Time Kernel

At the core of RROSS is the real-time Kernel which is made up of the following features:

- Multitasking process management
- Interprocess communication/synchronization

- Memory management
- Interrupt handling
- System Clock and Timer support

These features form the basis of the application execution environment (or runtime) and also serve as operating system primitives which can be used to build up further features and extensions.

2.2.1.1) Multitasking Process Management

The real-time application environment is made up of prioritized periodic processes (tasks) with aperiodic (asynchronous) events and is best scheduled with a preemptive priority model. The preemptive priority model will allow the user to support timelines using timer interrupts, single loop programs with selected or no interrupts, and a variety of combinations of priorities, interrupts, critical and non-critical tasks. Task context switch time will be constant and deterministic. The Multitasking environment will allow for task creation, deletion, and suspension. (Reference 3.2.1.1)

2.2.1.2) Interprocess Communication and Synchronization

The cooperating Multitasking processes require a set of mechanisms to share memory, transfer data/messages, and synchronize actions.

Shared memory is the most simplistic method for tasks to share data. Tasks can communicate by using pointers to the shared memory. This simplicity of task communication requires services to handle concurrent access to the shared memory. Methods to deal with this Mutual Exclusion include disabling interrupts, executing in a critical region (disabling preemption) and using semaphore locking techniques. In this last case, a priority inheritance technique will be available to handle the problem of priority inversion.

The basic form of data transfer between tasks or between an interrupt handler and a task will be through a Message Queue facility. This facility will support the ability to transfer variable length messages at the Kernel level. Additional support at a higher level (such as Pipes and Sockets) can also be offered for those applications that can incur the overhead involved in their use.

Task synchronization will be provided using semaphores. A task or interrupt handler will signal the occurrence of an event by giving the semaphore. The signaled or handling task will wait on the semaphore and become ready to execute when the semaphore is given. Task synchronization that requires an associated transfer of data is inherently provided by the Message Queue facility.(Reference 3.2.1.3)

2.2.1.3) Interrupt Handling

As interrupts are the basis of asynchronous events and require timely action, efficient interrupt handling is a critical Kernel function. Fast interrupt handling to reduce interrupt latency will be provided.

A service will be provided to install and uninstall application interrupt handlers to system interrupts not managed by the Operating System.

The bulk of interrupt processing should be sent to a multiprocess level task to provide prioritization, queuing and reduced interrupt latency. A set of interrupt to multiprocess communication services will be provided for this purpose.

The operating system should not prohibit the development of interrupt handlers which can handle nested interrupts on the same or different levels. Masking of interrupts on a level by level basis must be supported when supported by the hardware architecture. (Reference 3.2.1.9)

2.2.1.4) System Clock and Timer Support

A real-time clock will be provided to keep track of system time. Services will be provided to set, update and read the clock. The ability for a task to be signaled after a specified time delay will be provided. The handling of this time-out will be done at task level. The ability to invoke an application interrupt handler by the delay timer will be provided.

Due to power conservation measures in spacecraft, the processor clocks are often turned down to lower frequencies. This is done while cruising to the destination or waiting for events of interest to occur. The clock services must be tailorable for changing in the hardware clock rate. The operating system must be able to dynamically control the interpretation of the clock increment rate. If the hardware architecture has the time of day clock tied to the system clock, then changing the processor changes the TOD clock. When using the time of day user programs and system software must get the correct time based on the new clock rate. (Reference 3.2.1.8)

2.2.1.5) Memory Management

The basic Kernel requires a memory management function which allows dynamic memory allocation and deallocation from a fixed block memory pool. This function will be used by any program component to allocate and free memory, this includes the application as well as any operating system component or extension including the Kernel.

The size of the memory pool in both total size and block size will be configurable by the user. (Reference 3.2.1.2)

2.2.2) Virtual Memory Management

Virtual memory management should be optionally available. Besides allowing memory to be extended to other devices which is of limited use to a spacecraft system, virtual memory management allows memory protection and remapping. Memory protection should include write protection on a block by block basis where the size of the block is determined by the system implementation. Protection may be offered on a memory block such that only certain users may access it. Memory mapping should allow a block of memory to *replace* another. When failed memory cells are detected in a block, it is desirable to disable that block and enable a new memory block which will respond to the same virtual (user) address. The replacement memory would be kept in reserve and the amount of reserve should be configurable when the user programs are compiled and linked or when the system is initializing on the embedded processor. (Reference 3.2.1.2)

2.2.2.1) Data Redundancy

Memory allocation and access procedures should be provided to allow user programs to allocate memory which will be duplicated and updated in a controlled way such that one copy of the data is always consistent even if the program stops while updating it. A function should also be provided which would evaluate the consistency of the data and *fix* any inconsistent copies using the other copy as a reference. The functions required would include versions of: allocate, deallocate, read, write, and clean. This memory space must not be returned to the available memory pool when the operating system restarts after an error. It must be kept reserved and owned by the program which created it so that when the owner is restarted, the data is still available. If program ID's may not be consistent between OS initializations, then a user name or user defined identifier must be used to reserve the memory space.

An additional function may be provided to allocate memory space which will be saved for the owning program after a system restart, but which is not duplicated. It would be up to the user program to guarantee consistency of the data contained in the space.

A function should be provided which reports the ownership of a block of *reserved* memory. Another function should report all blocks owned by a specified program. These functions will help in cleaning up space if system restarts are not successful. Another function should be provided which will clean all reserved memory spaces so that an operating system initialization may reinitialize everything and start over from scratch. (Reference 3.5.2.1)

2.2.3) File System

RROSS will have the capability to interface to File Systems which support the storage of groups of data in named files. The basic I/O primitives for these files will be routed by the I/O System to the appropriate File System for processing. The File System will contain the function needed to implement the particular file format supported and drivers for supported devices.

File Systems can be provided and added as necessary to handle any file format and device needed. While it is doubtful a disk device File System would be used during a space mission, it is possible one may be included during development for load capability or other use.

RROSS should provide a memory based File System which provides the capability to read and write collections of data in named files. File support for a sequential recording device must not be precluded. (Reference 3.2.1.7)

2.2.4) I/O System

The RROSS I/O System will present a standard, device independent view of any I/O device. A device is an object which is used to send data to and receive data from. The type of device determines the source and destination of this data. A device can be a physical device such as a:

- Communication port
- Bus or Backplane interface (e.g. 1553 or VME)
- Discrete line
- Interface to custom hardware
- Disk drive

or a logical device such as a:

- Pipe (an intertask message I/O device)
- Socket (an interface of the Network System)
- RAM Disk drive

To enhance portability and reusability the user interface will be compatible with the UNIX I/O system. The interface implements a set of the seven basic I/O primitives:

1. create
2. remove
3. open
4. close
5. read
6. write
7. ioctl

where the primitives used and their implementation specifics differ according to the target device. The seven basic I/O primitives operate on named objects called files. A file can be an actual file contained on a file system whether local or remote, or it can be a named object that refers to an I/O device. I/O is accomplished by reading from and writing to the named file regardless of the physical implementation of the I/O device. The one exception is for network communications in the form of Sockets which are not described by named files. Creating a socket does return a file descriptor which can be used with standard I/O system reads and writes. The Network System is described in section 3.2.1.6. In the case of actual files on a file system, the basic I/O primitives are passed to the File System component which manages the I/O requests. The File System is described in section 3.2.1.7. The relationship between the I/O System, Network System, and File System is depicted in Figure 1. (Reference 3.2.1.4)For I/O requests of devices other than the File System or Network, the basic I/O primitives are implemented in device drivers for the specific device. One function of the I/O System is to route the I/O primitives to the File System, Network System, or specific device driver as appropriate. The I/O System will also provide the capability to dynamically install device drivers. This allows the I/O system to be extended to handle any device as needed. (Reference 3.2.1.5)

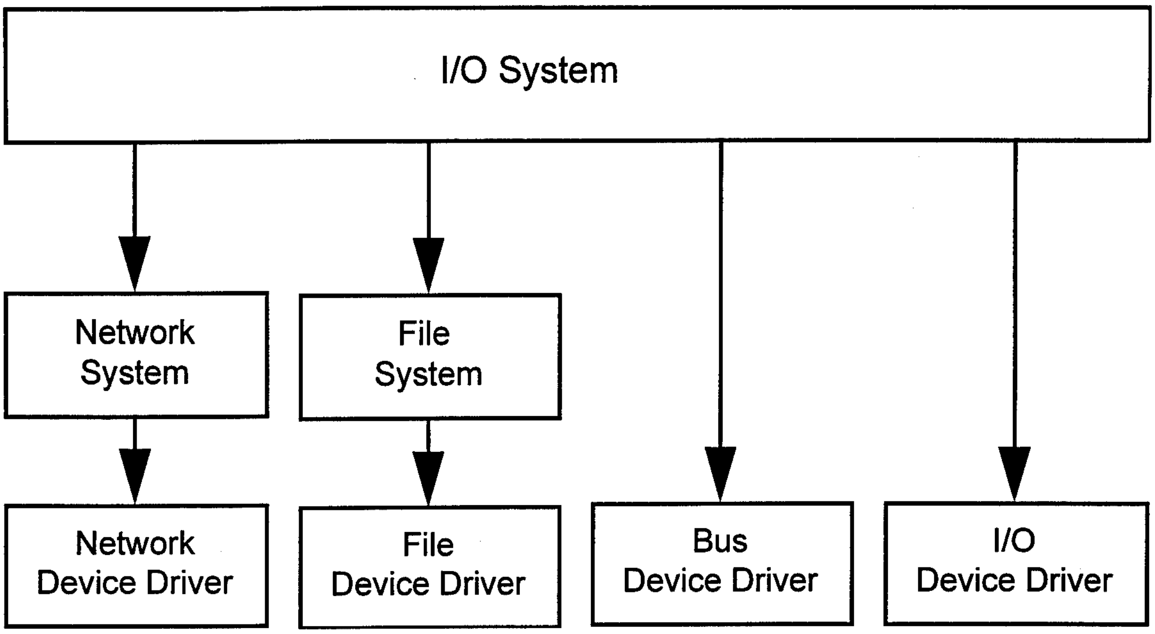


Figure 1. I/O System

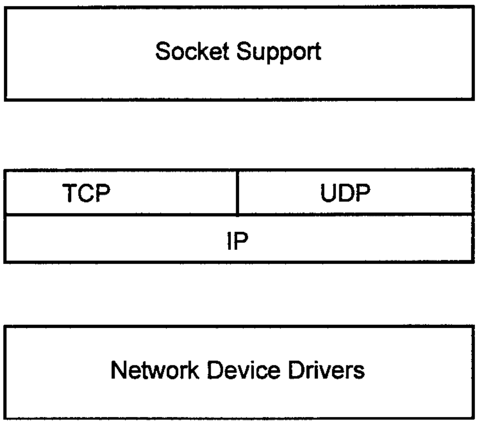


Figure 2. Network System

2.2.5) Network System

Network support provides the connectivity needed for an Open Systems environment. By utilizing the industry standard Internet Protocol (IP) for network communication, RROSS can communicate with any connected system which supports the IP standard.

RROSS will extend the Internet Protocol with the User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) for process to process communication. Both of these protocols extend the IP address with a port address; TCP adds a layer of reliability and guaranteed data transmission with a point to point connection between processes.

The application interface to Network communications and the standard protocols will be provided by the Socket abstraction. A Socket is an endpoint for communication and can be created for datagram use (UDP), or stream use (TCP). Using Sockets for communication provides a level of open connectivity, processes in RROSS can communicate with any other process which uses the Socket abstraction. The other process can be another RROSS process or a process running under a different OS which supports the Socket interface. Additionally, the communication is the same regardless of where the two processes exist, they can be running on the same CPU, on different CPU's sharing a common backplane, or on different CPU's connected by a network. This adds to the portability of an application as it can be distributed to run as needed.

At the lowest level are the Network Device Drivers which handle the physical transmission implementation for the target medium. Device drivers can be provided which handle standard network devices such as Ethernet or SLIP. Device drivers can also be added to handle network communications over the system backplane or other processor interconnection. This allows the high level network facilities to be used no matter how the processors are connected.

The Socket interface to network communications can be used to build up other OS functions and standard interfaces such as:

- File Transfer Protocol (FTP)
- Remote Procedure Calls (RPC)
- Telnet
- Network File System (NFS)
- OS Bootstrap Loader
- Application Loader
- Remote Debugger

RROSS will support multiprocessing with the Network System, processes will communicate using the standard network functions. This allows physical networks to be changed and applications to be moved among processors without changing application code. The underlying communication implementation (from TCP down to the basic driver level) can be chosen to provide the level of openness and performance desired. (Reference 3.2.1.6)

2.2.6) Operating System Load

The operating system should be able to execute from ROM while using RAM only for writable data. Since all processors have some RAM available, the operating system must support being loaded into RAM. The OS may load itself from ROM, or it may rely on a boot program to load it and start its execution. Users must be allowed to rewrite any boot programs since the operating system may be loaded from ROM, a communication device, or placed into RAM by an independent device using DMA access to local memory. DMA access requires hardware support. (Reference 3.2.1.10)

2.2.7) Application Load, Start, and Stop

User programs must be allowed to load from many sources. They may be kept in ROM or NVRAM (some type of memory) or they may be kept elsewhere in the spacecraft or on the ground (requiring a communication channel for loading). Programs should be loadable from a file (a named image) kept on any device.

Loading an application (user) program may be initiated by the operating system or by another user program which will then become the parent of the new program, the child.

Application programs must be allowed to kill other programs or themselves. When an application program is killed, the memory space and any other resources it used must be reclaimed by the system.

To guard against programs being killed erroneously, programs that initiate kill commands should use an independent means to check the need to kill a program. An example implementation might be code which builds the command message, verifies from some separate data or message the need to kill the program, and then sends the message. (Reference 3.2.1.10)

2.2.8) Error Management

Errors must be propagated upward through subroutine calls and child-parent relationships until they can be handled by a user program. A user program may elect not to handle a reported error, but at least one user program must be given the opportunity. Errors in subroutines must be passed back to the caller; errors which cause programs to fail (crash) must be reported to the parent of the failed program. Optionally, users may be allowed to monitor errors of programs, communications, or system services.

Ada exceptions behave as other errors. Exceptions must be propagated from subroutine to caller, and from child task to parent. Exception handlers may use interprocess communications to report errors to other programs or tasks. (Reference 3.5.3.1)

2.2.9) Monitoring and Logging

User programs and system services should be able to record the fact that they executed, in what order they executed, parameters passed in or out, and whether or not they were successful. The record is invaluable for debugging problems with the interaction of multiple programs. Monitoring may be provided with procedures which control access to a shared memory space and record the event in a consistent manner. The biggest disadvantage is that the shared log becomes a critical resource required by all programs, but accessible by only one at a time. Monitoring may be provided by a another task or program executing asynchronously to other user programs. Messages are sent to the monitor task using standard interprocess message facilities. The messages should optionally be logged asynchronously without impacting the performance of the caller, or synchronously which requires the caller to wait until the entry has been made in the log. The monitor may provide performance information. (Reference 3.2.2.3)

2.2.10) Error Recovery

2.2.10.1) Error Detection

The operating system must support hardware interrupts caused by error detection mechanisms. The operating system must allow the user programs to access any status registers or other error indicators.

The operating system services should always provide the standard error checking protocols on communication channels and messages. The standards will differ according to device and protocol. Storage of data should be optionally protected using checksums, parity, or other means.

The operating system must allow the user programs to define and implement communication and storage protocols to protect their critical data. The operating system or user programs must be allowed to *scrub* memory periodically where memory error correction is supported.

The operating system or user programs may implement time-outs or *deadman* routines to detect the premature demise of a task executing in the system. These health checks must be supported between tasks on different processors when implemented in a multiprocessor system. (Reference 3.5.3.2)

2.2.10.2) Restart

The operating system must be able to restart without being reloaded to memory. A initialization process must initialize any state information required to be in a predetermined state at startup. In addition a user defined initialization routine must be allowed to be inserted into the startup process. This will allow any user defined state data to be placed into a consistent state before use.

Being able to restart the system and the user programs without reinitializing the entire memory space is the key to RROSS error recovery. Programs will need state information and other data to reconstruct and resume activities interrupted by errors or hardware failures.

The operating system should support program restarts initiated by user programs. A user should be able to restart the operating system itself, a user program (including itself), or a family of user programs (a user program, its children, and siblings). (Reference 3.5.2.2)

During startup, only that data which is required by the configured services should be initialized. Those services not chosen should not slow startup by initializing state data. Configurability of initialization should allow the user to specify which services are critical to the system so that only those critical services are initialized before the user programs are executed. After the user programs have been given control, they would then allow initialization of secondary services in the idle or spare time, or specify when initialization would occur.

When hardware errors are detected during the startup process and redundant assets are available, replacements should be made and diagnostics should be run on the failing assets in idle or spare time. The failed assets may be reported to user programs after the startup sequence is complete. (Reference 3.5.2.3)

2.2.10.3) Checkpoint

When the system restarts, the user programs should be allowed to begin execution close to where it was executing before the restart was initiated. To accomplish this, the user program must be allowed to store its state data as it is executing and retrieve it when it restarts. The operating system may support this function by:

- periodically saving all state data for all active tasks
- by providing a checkpoint function which saves all state data when directed by a user program
- by providing safe storage routines which allow user programs to save their own state data without fear of data inconsistency
- by allowing the user programs to save their own state data without fear of the operating system reclaiming the space and initializing it.

(Reference 3.5.2.2)

2.2.10.4) Data Recovery

Data recovery must be allowed after an error has been detected. The operating system must not prohibit storing data redundantly or using checksums or other encoding algorithms. Also, the operating system must allow storage to be reserved so that it will not be reclaimed and initialized when restarting the system. The operating system may optionally provide services to facilitate these capabilities. (Reference 3.5.2.1)

2.2.10.5) Program Recovery

Programs must be allowed to start close to the point in the code executing when the error occurred. The operating system may directly start the program at that point if it knows where it is, or the program may be allowed to start from the beginning again. It would be up to the user program to access state data stored before the error to determine from where to branch to continue executing. (Reference 3.5.2.2)

2.2.10.6) Communication Recovery

Programs must be allowed to request and receive acknowledgment for transmissions and interprocess messages. The state data associated with a program may contain the status of expected versus received acknowledgments. Upon restarting, the unacknowledged messages may be resent. Receiving programs would be expected to ignore receipt of multiple, identical messages if they would be harmful. (Reference 3.2.2.2)

2.2.10.7) Hardware Recovery

Hardware failures are generally difficult problems to solve with software. Some errors can be effectively handled however. Memory failures can be handled by reserving a portion of the physical memory to be used later as replacement memory for a failed block. The operating system and the hardware architecture must provide memory remapping to support this recovery action. Communication device failures may be handled by allowing a spare or unused device to be allocated to replace the failed device. The user programs may change devices being used or the operating system may rename the unused device to appear as the failed device, thus allowing the user program to remain ignorant of the actual device used.

Where the system hardware allows, the operating system must support, or allow user programs to support, replacement of hardware assets. Processors, sensors, power supplies, memory, or other assets may be brought on-line, reloaded, or restarted and processing must continue from a predetermined point. (Reference 3.5.2.3)

2.3) User Characteristics

The users of this system are software developers who are most concerned about data integrity, size of system and user software, and the predictability of scheduling user programs. Since these systems often control the navigation, guidance, and maneuvering of a spacecraft, control data must always be correct and timely. Since radiation hardened memory and processors are very expensive, the size and complexity of the system software and compiled user software is a concern. Spacecraft memories are typically small, 64KB to 64MB, and processors are often slow, 1 MIPS to 50 MIPS. Because guidance and instrument deployment use strict timelines, it is imperative that the scheduling of the user programs which control spacecraft functions be entirely predictable and that dispatching within time frames can be guaranteed. Error detection and recovery mechanisms are a key to guaranteeing predictable behavior. These features most strongly influence spacecraft software and, therefore, this specification. (Reference 3.4.2)

2.4) Constraints

Additional constraints include migration in the space community away from assembly code toward higher level languages such as C or Ada. Due to the expense of developing, launching, and monitoring a spacecraft, mission success is all important. The operating system is always considered *mission critical* meaning that some or all goals of the spacecraft will not be obtainable if this component fails. The validation of all mission critical components is a strict and detailed process. The operating system must be thoroughly tested and detailed test reports should be available. Error injection mechanisms must be available for testing and validation of the user's system. In addition, most user groups will request source code be provided so that system problems can be addressed at all levels including hardware internals and software instructions. (Reference 3.5.1 and 3.5.4)

2.5) Assumptions and Dependencies

2.5.1) Architectural Guidelines

These guidelines refer to the system architecture of which the operating system is only a part. These are guidelines, not requirements and are intended to provide context and boundaries for the operating system design.

2.5.1.1) Performance

Performance must be provided such that all tasks identified as critical can be completed within required time periods. These tasks include the user programs themselves, operating system overhead to execute them, and the execution of any program or interrupt handler which may preempt them.

Operating system overhead includes system calls, context switching, memory management, saving state data at a checkpoint, and coordinating threads or programs on multiple processors.

The size and complexity of the user programs will dictate the majority of the processor performance required by software. That performance may be measured in any way, such as MIPS or time to complete a standard program. A margin for growth should be specified to accommodate changes in design and peaks in processing needs, i.e. Analysis shows *x* MIPS is required, so 1.2 * *x* MIPS is specified to allow a 20% margin. A margin of 25 % should be provided. (Reference 3.4.2)

The second major factor in determining processor performance requirements is power usage. A spacecraft has severe power limitations; batteries are very heavy, nuclear fuel is expensive, and solar radiation is not available in shadows and is weak when far from the sun. Power is usually the most precious resource in spacecraft electronics systems. By using very slow clocks for a processor, the power required by that processor can be minimized. A balance must be made between the needs of software and the limits of the power subsystem. (Reference 3.2.2.1)

One or many processors can be used to provide this performance. Another reason to use multiple processors is to support error recovery. When recovery from an error, including a permanent critical fault, must be done more quickly than a spare processor or subsystem can be brought on-line, then a hot spare must be available. A hot spare is running the same programs as the primary unit and can monitor and take over for the primary unit immediately upon detecting an error in the primary unit. Spares may also be warm, meaning they are powered and loaded, but not executing user programs; or cold meaning not powered or loaded. Warm and cold spares provide redundancy without using as much power while sacrificing recovery time. Cold spares are used when error recovery need not be done quickly since loading usually takes a relatively long time.

The operating system must be able to accommodate any of the above types of spare processors. This support may be provided by customized user services or programs. Communication channels must be available with the same privileges already assigned to the original primary unit. Shared data areas must be recoverable by going back to a previous state and reopening access even if the primary unit had it locked when it failed. Busmaster-ship, when applicable, must be transferable to a working unit. Memory areas and communications identifiers should be remappable to allow replacement of a processor without affecting identifiers used by user programs. (Reference 3.2.3.1)

In distributing user programs across multiple processors, the operating system may take the following into account: processors specified by user programs, current CPU usage, operability of a processor and its local resources and the availability of spares for it.

2.5.1.2) Storage

Storage requirements are affected most severely by the following factors: available space, power usage, software size, and buffering of image data during and after peak sensor use. The amount of storage specified must be a balance of available system resources versus software needs.

A margin for growth should be specified to accommodate changes in design and peaks in storage needs, i.e. Analysis shows *x* KB is required, so 1.2 * *x* KB is specified to allow a 20% margin. A margin of at least 50% should be provided.

Type of storage must also be considered. RAM will be used for dynamic data and probably programs, but NVRAM and ROM are also available. More robust storage such as NVRAM can be used to store state data which is relatively stable and would aid initialization after power is applied to the system in the future. ROM can be used to store initialization programs and data to be used when the system is powered on for the first time or subsequent times. Specialized routines are often required to access non-standard storage media. The operating system must not deny the user the freedom to create user defined device drivers to access any medium. (Reference 3.4.2)

Error recovery requirements may increase the required system storage. In order to ensure the state of data, multiple copies may be maintained by the operating system or user programs. The multiple copies should be physically separated as much as possible, e.g. separate memory boards with separate power supplies. Extra storage may be used as hot spares, meaning data is actively updated on the duplicate copy by software. Extra storage may be used as cold spares. Cold spare storage must be brought on-line and initialized before it is available for use. What type of spare storage, if any, is required is a balance of power usage, available space, and error recovery needs.

2.6) Software Structure and Performance

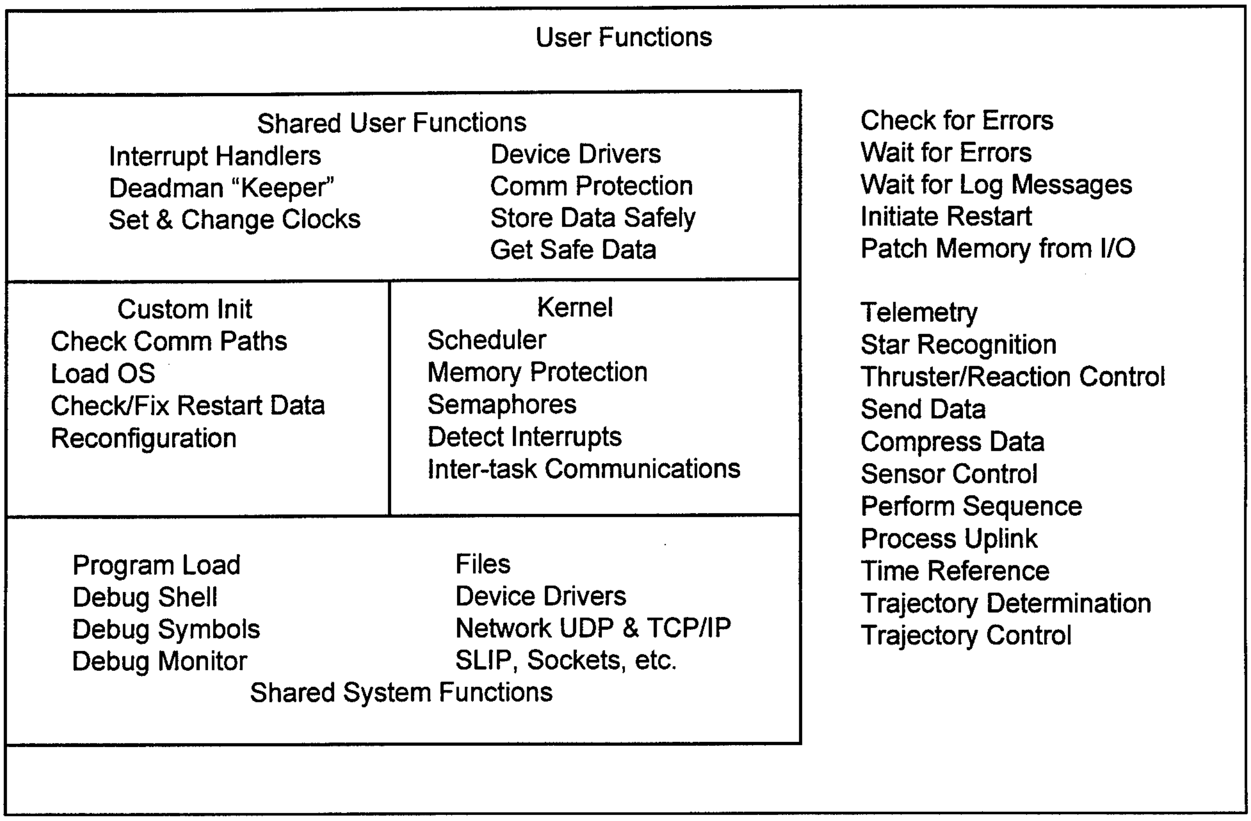
A modular architecture is desirable to minimize the effort to implement the operating system on a new hardware architecture. Hardware dependent pieces of code should be small and collected together. (Reference 3.5.5)

The ability to customize the operating system by creating user defined extensions to add new function or replace predefined system functions allows maximum flexibility. The system implementation may create interfaces for optional functions which the users can implement as they desire.

Extended capabilities may be provided by system software or the user. The system software provided must not prohibit the user from providing customized extensions. Any constraints on such customized functions or user extensions must be documented clearly to facilitate their development. (Reference 3.5.6)

The partitioning of the operating system extensions between the kernel, application program interfaces (API), or application programs (AP) is left to the developers. RROSS is specified to allow users to customize, create, or remove system features by writing services and application interfaces for those services. Since a service written by a user may be called as efficiently as services provided by the kernel or system libraries, there is no expected performance difference based on the origin or location of the services.

While performance should not be dependent upon the partitioning of the software functions, convenience will be. For usability, it is recommended that shared functions such as those related to timers, communications, saving data, retrieving data, loading software, and monitoring deadman timers be placed in shared libraries whether developed by the operating system vendor or the user. Functions such as reconfiguration and custom initialization should be implemented as part of the operating system initialization sequence. This is usually supported as a compile and link customization service. All other programs should be considered autonomous user programs which use services of other programs and provide services in return. While this is practically no different from the other functions, conceptually, the remaining user functions are only used by a few specific higher level functions and are not generally used by most programs in the system.



User Functions

Shared User Functions

Interrupt Handlers
Deadman "Keeper"
Set & Change Clocks

Device Drivers
Comm Protection
Store Data Safely
Get Safe Data

Custom Init
Check Comm Paths
Load OS
Check/Fix Restart Data
Reconfiguration

Kernel
Scheduler
Memory Protection
Semaphores
Detect Interrupts
Inter-task Communications

Program Load
Debug Shell
Debug Symbols
Debug Monitor

Files
Device Drivers
Network UDP & TCP/IP
SLIP, Sockets, etc.

Shared System Functions

Check for Errors
Wait for Errors
Wait for Log Messages
Initiate Restart
Patch Memory from I/O

Telemetry
Star Recognition
Thruster/Reaction Control
Send Data
Compress Data
Sensor Control
Perform Sequence
Process Uplink
Time Reference
Trajectory Determination
Trajectory Control

Figure 3. Software Partitioning

3.0) SPECIFIC REQUIREMENTS

Each requirement is of the form:
3.n.m) The operating system shall ....

where Necessity is expressed as:
shall = essential - required to meet specification
should = optional - desired but exceeds requirements

3.1) External Interfaces

3.1.1) Development Interface

1	RROSS development tools shall provide at least one compiler and a linker to create loadable, executable image files from Ada or C source files.
2	RROSS development tools should support multiple host computers.
3	RROSS development tools should support multiple developers working in parallel on the same multi-user host computer.
4	RROSS development tools shall produce program images which can be loaded into the target processor's memory without manual manipulation of the image files.
5	RROSS tools should allow the use of any editor supported by the host system in place of the tools' default editor.
6	RROSS tools should provide a simulator which can execute user programs on the host computer as they would run on the target computer.
7	RROSS tools should provide a simulator which supports source level symbolic debug and instruction level visibility.
8	RROSS simulation tools should provide the ability to add programmed responses to stimuli such that hardware attached to the processor can be simulated, e.g. extended memory access functions by an MMU or accessing registers in an I/O device.
9	RROSS tools should provide a profiler which can analyze the performance and data flow of a user program.
10	The RROSS profiler should provide the timing for a user program such that the execution time for each instruction or block of instructions is estimated and reported.
11	RROSS tools should provide the ability to control user source code such that dependencies between files are known and recompiling one file automatically compiles dependent files.
12	RROSS tools should provide the ability to control user source code such that different versions of files referenced by version number can be compiled to create multiple versions of executable code.
13	RROSS tools should provide the ability to control user source code such that only one user may update a file at a given time.

3.1.2) Debug Interface

14	RROSS tools shall provide an interface to load the operating system (RROSS) into the target processor's memory.
15	RROSS shall provide an interface to load user programs into the target processor's memory.
16	RROSS tools shall support an interactive interface to start and stop user programs.
17	RROSS tools shall support an interactive interface to query the status of all tasks, e.g. running, blocked, defunct.
18	RROSS tools shall support an interactive interface to read and modify memory referenced by address.
19	RROSS tools shall support an interactive interface to read and modify variables used by RROSS or user programs.
20	RROSS tools should support an interactive interface to read and modify interprocess communications data structures.
21	RROSS tools should support an interactive interface to read and modify interprocessor communications data structures.
22	RROSS tools shall provide maps which contain the addresses of memory blocks which have been statically allocated for user programs, system services, variables, stacks, or other uses.
23	RROSS shall provide, upon query, the addresses of memory blocks which have been dynamically allocated for user programs, system services, variables, stacks, or other uses.
24	RROSS debug tools should provide conditional breakpoints.
25	RROSS debug tools should provide the ability to perform preprogrammed responses to reaching a breakpoint.

3.2) Functions

3.2.1) Standard OS Features

3.2.1.1) Scheduling

26	RROSS shall provide a preemptive priority scheduling algorithm.
27	RROSS shall provide at least 32 levels of priority for user tasks.
28	RROSS shall provide multitasking functions including creation, deletion, and suspension of tasks.
29	RROSS shall provide priority inheritance for any resource, including semaphores, which is shared between multiple tasks.
30	RROSS shall provide deterministic context switching between tasks in bounded time, including Ada tasks.

31	RROSS shall provide documentation which specifies the rules governing context switching between tasks, including how to predict which task should be running after the switch.
32	RROSS shall guarantee that a task with the highest user priority and a periodic timer set for 8ms intervals should be allowed to run at the 10ms intervals within 1ms.
33	RROSS shall guarantee that a task with the highest user priority and which is attached to an interrupt should be allowed to run within 1ms of the processor's report of the interrupt.
34	RROSS shall provide services to allow a user program to enter a critical section of code which may not be interrupted or suspended until the critical section is exited.
35	RROSS should provide services to dynamically set the priority of a user task in order to suspend all other tasks when a high priority situation is found by a low priority program, such as a background check finding an error.

3.2.1.2) Data Management

36	RROSS shall provide software interfaces for dynamic allocation and deallocation of system memory.
37	RROSS shall provide memory management services to dynamically allocate and free memory from a fixed block memory pool or an Ada heap.
38	The size of the memory pool used for dynamic memory allocation and the size of the blocks defined within it should be configurable by the user.
39	RROSS should provide protection on a block by block basis against writing memory by unauthorized user programs.
40	RROSS should provide virtual memory services which reroute access of a specified block of memory to another specified block of memory.
41	RROSS should allow specified blocks of memory to be hidden from use such that they can be used as replacement blocks when calling the memory rerouting services.

3.2.1.3) Interprocess Communication

42	RROSS shall provide the capability for multiple tasks to share memory and coordinate access to the shared memory.
43	RROSS shall provide message queue facilities which support variable length messages at the kernel's priority level.
44	RROSS shall provide message queue facilities which support messages from interrupt handlers to user programs.
45	RROSS shall provide semaphores to be shared between tasks and services to access the semaphores atomically.
46	The RROSS message queue facility shall provide variable sized messages and data packets.

47	RROSS should provide to the sending program acknowledgment of receipt of interprocess messages.

3.2.1.4) I/O Facilities

48	The RROSS I/O primitives create, remove, open, close, read, write, and ioctl shall provide user interfaces which are independent of the resource being accessed with the exception of network sockets.
49	RROSS shall provide UNIX I/O primitives referenced by a named object and using the primitive **create** to return a file descriptor for a resource.
50	RROSS shall provide UNIX I/O primitives referenced by a named object and using the primitive **remove** to free the file descriptor for a resource.
51	RROSS shall provide UNIX I/O primitives referenced by a named object and using the primitive **open** to establish an input or output path with a resource.
52	RROSS shall provide UNIX I/O primitives referenced by a named object and using the primitive **close** to close an input or output path with a resource and place the resource into a state ready to be opened again.
53	RROSS shall provide UNIX I/O primitives referenced by a named object and using the primitive **read** to read input from the resource to the user program.
54	RROSS shall provide UNIX I/O primitives referenced by a named object using the primitive **write** to write output to the resource from the user program.
55	RROSS shall provide UNIX I/O primitives referenced by a named object and using the primitive **ioctl** to perform specialized functions unique to the resource.

3.2.1.5) Device Drivers

56	RROSS shall provide device drivers referenced by a named object.
57	RROSS device drivers shall support the I/O primitives create, remove, open, close, read, write, and ioctl.
58	RROSS shall provide user interfaces for communications devices and allow device drivers to be defined and implemented by the user.
59	RROSS shall provide UNIX device drivers referenced by a named object and using the primitive **create** to attach the driver to a device and return a file descriptor for the driver.
60	RROSS shall provide UNIX device drivers referenced by a named object and using the primitive **remove** to unattach the driver from a device and free the file descriptor for the driver.
61	RROSS shall provide UNIX device drivers referenced by a named object and using the primitive **open** to establish communication with the device as an input or output device.
62	RROSS shall provide UNIX device drivers referenced by a named object and using the primitive **close** to close communication with the device and place the device and driver into a state ready to be opened again.

63	RROSS shall provide the UNIX device driver primitive **close** such that the device and driver are placed into a state ready to be opened again following a communications or device error.
64	RROSS shall provide UNIX device drivers referenced by a named object and using the primitive **read** to read input from the device to the user program.
65	RROSS shall provide UNIX device drivers referenced by a named object using the primitive **write** to write output to the device from the user program.
66	RROSS shall provide UNIX device drivers referenced by a named object and using the primitive **ioctl** to perform specialized functions unique to the device.

3.2.1.6) Network Connectivity

67	RROSS shall provide support for the Internet Protocol (IP).
68	RROSS shall provide support for the User Datagram Protocol (UDP).
69	RROSS shall provide support for the Transmission Control Protocol (TCP).
70	RROSS shall provide sockets compatible with BSD 4.3 UNIX sockets.
71	RROSS should provide File Transfer Protocol (FTP).
72	RROSS should provide Remote Procedure Calls (RPC).
73	RROSS should provide Telnet remote logon.
74	RROSS should provide Network File System (NFS).
75	RROSS should provide application loading across a network using IP.
76	RROSS should provide a remote debugger which connects the host and the target computer across a network using IP.
77	RROSS should allow the user to provide an operating system loader based upon IP for loading across a network.

3.2.1.7) File System

78	The RROSS file system shall support the I/O primitives create, remove, open, close, read, write, and ioctl.
79	RROSS shall provide a file system which supports system memory as a file media where a file refers to a named collection of data.
80	RROSS should provide a file system which supports a disk device as a file media.
81	RROSS shall allow the user to provide a file system which supports any readable and writable storage media.
82	RROSS shall allow the user to provide a file system which supports any readable storage media, e.g. Read Only Memory (ROM).

3.2.1.8) Clocks and Timers

83	RROSS shall provide user interfaces for accessing at least the following clocks: a time-of-day clock, a delay timer which causes an interrupt, and a periodic timer which causes interrupts at regular intervals.
84	RROSS shall provide a time-of-day clock which shall keep a reference time from an arbitrary date.
85	RROSS shall provide services to set and read the time-of-day clock.
86	RROSS shall provide services to set a delay timer which should signal the requesting task after the specified delay.
87	RROSS shall provide services to set a periodic timer which should signal the requesting task regularly at the specified interval.
88	RROSS shall support a granularity of at least 1ms per tick on the time-of-day clock and all timers used for delays and periodic signals.

3.2.1.9) Interrupts

89	RROSS shall provide services to install and remove user defined interrupt handlers for all interrupts provided by the hardware.
90	RROSS shall not preclude the development of interrupt handlers which handle nested interrupts on the same or different levels.
91	RROSS shall provide services to mask interrupts on a level by level basis when such masking is supported by the hardware architecture.

3.2.1.10) Loading and Unloading Software

92	RROSS shall allow the user to provide an operating system loader which may load RROSS from any storage or communication media.
93	RROSS shall provide application loading from named files stored on any media which is supported by the file system.
94	RROSS shall support application loading being initiated from the operating system or another application (user program) which should become the parent of the loaded program (child).
95	RROSS shall support user programs killing or removing themselves and other programs.
96	RROSS shall free all resources allocated to a program when that program is killed or removed.

3.2.2) OS Extensions for Space Development

3.2.2.1) Dynamic Clock Rates

97	RROSS shall provide a method to dynamically control the interpretation of the clock increment rate, i.e. user software may control the system clocks such that one clock tick may represent 1ns at one system frequency and 8ns at another frequency and the time-of-day clock and other timers must be adjusted to maintain correct orientation.

3.2.2.2) Communication Safeguards

98	RROSS device drivers shall report any detected errors to the user program calling the driver.
99	RROSS should provide a service to report errors to any task, including a task other than the one which opened the device.
100	RROSS shall provide UNIX device drivers such that by using the **remove** and **create** primitives, the user can attach the driver of a failed device to another equivalent device.
101	RROSS should provide interfaces such that multiple processes receive the same **read** data from a device, i.e. two programs read same device and are given the same data in the same order.
102	RROSS should provide interfaces such that multiple processes may write to one device and the data written is compared and must match to be transmitted; mismatched data is reported to calling processes.
103	RROSS should provide interfaces such that data written to a device is transmitted on multiple hardware interfaces simultaneously.
104	RROSS should provide interfaces such that data received from multiple hardware interfaces is compared before being supplied to a user program performing a **read** from the device; data errors are reported to reading processes.

3.2.2.3) System Visibility

105	The RROSS kernel and service routines shall have the ability to write to a common log such that the calling order of the services is preserved for later retrieval.
106	The RROSS kernel and service routines should have the ability to write to a common log asynchronously by sending message to separate logging task or synchronously by using a subroutine or macro to access the log.
107	The RROSS kernel and service routines should write to a common log the identity of executing service, parameters passed in and out, and success or failure status.
108	RROSS shall provide a software interface to user programs to log messages in a shared area with the characteristics of a circular queue for later retrieval.
109	RROSS shall provide a software interface to user programs to log messages in a shared area of at least 1KB in size.
110	RROSS shall provide the means for a user program to patch the program image or data belonging to itself or other user programs.
111	RROSS should provide a service to report which tasks have opened a specified device driver.
112	RROSS should provide a service to report which device drivers have been opened by a specified task.
113	RROSS device drivers should log any detected errors for later retrieval.

114	RROSS shall provide an interface to user programs to load other user programs into the target processor's memory.
115	RROSS shall provide an interface to user programs to start and stop other user programs.
116	RROSS shall provide an interface to user programs to query the status of all tasks, e.g. running, blocked, defunct.
117	RROSS shall provide an interface to user programs to read and modify variables used by RROSS or other user programs.
118	RROSS should provide an interface to user programs to read and modify interprocess communications data structures.
119	RROSS shall provide an interface to user programs to read and modify memory referenced by address.
120	RROSS shall allow user programs to access any and all registers which the hardware allows to be read or written.

3.2.3) OS Extensions for Distributed Processing

3.2.3.1) Interprocessor Communications

121	When implemented for a multiprocessor system, RROSS shall provide shared memory, semaphores, and message queue facilities such that the user is not required to know which processor is executing any program.
122	When implemented for a multiprocessor system, RROSS should allow the retrieval of program state data saved by tasks on one processor by tasks on another processor.
123	When implemented for a multiprocessor system, RROSS should provide distribution of tasks across processors based upon requests of user programs, availability of processors, and current CPU usage.

3.3) Performance

124	When RROSS is loaded and the RROSS initialization code is invoked, RROSS shall reach an operational state and be ready to start user programs within one second.

Some requirements in other sections may refer to performance. These other references are located in specific sections due to their relation to the subject matter in those sections.

RROSS suppliers and developers are subject to all requirements in this specification regardless of the section in which they appear.

3.4) Design Constraints

3.4.1) Standards Compliance

3.4.1.1) Programming Standards

125	RROSS software interfaces for C shall conform with ANSI-Programming Language C,

	X3.159-1989.
126	RROSS software interfaces for C should conform with POSIX standard IEEE-Std-1003.1.
127	RROSS software interfaces for C should conform with POSIX standard IEEE-Std-1003.4 for real-time extensions.
128	RROSS software interfaces for Ada should be provided by POSIX Ada bindings as defined by IEEE-Std-1003.5.
129	RROSS software interfaces for Ada shall conform with the Ada language reference MIL-Std-1815A-1983.
130	RROSS software interfaces for Ada should conform with updates to the Ada language reference MIL-Std-1815, ANSI/ISO/IEC-8652:95 as they become the Ada standard.

3.4.1.2) Language Support

131	RROSS tools shall provide compilers for at least the following languages: Ada, C, Assembler.
132	RROSS shall support user program development using Assembler and either C or Ada.
133	RROSS should support user program development using Assembler, C, and Ada.
134	RROSS tools shall allow code written in Assembler to be called from C or Ada programs.
135	RROSS tools shall allow code written in C to be called from Ada programs.
136	RROSS tools should allow code written in C to be called from Assembler programs.
137	RROSS tools should allow code written in Ada to be called from C and Assembler programs.
138	RROSS shall provide documentation of the compilers' stack, register, and memory usage.
139	RROSS shall provide standard library services including numeric, string, bit manipulation, and trigonometric functions.

3.4.2) Constraints from System

140	There shall exist a minimum executable version of the RROSS kernel such that the program image displaces no more than 64KB of memory and requires no more than another 64KB of RAM memory.
141	There shall exist a version of the RROSS kernel such that the program image can be placed in Read-Only-Memory (ROM) and executed.
142	There shall exist a version of the RROSS kernel such that the program image can be placed in Random-Access-Memory (RAM) and executed.
143	RROSS shall not preclude the use of a CPU test interface such as the IEEE-1149.1 or IEEE-488.

144	RROSS should assume that the processor's maximum performance is three MIPS for worst case timing analysis (actual processor performances may be greater).

3.5) Software System Attributes

3.5.1) Reliability

145	RROSS shall provide detailed and complete documentation of testing and analysis performed to validate the operating system.
146	RROSS should provide detailed and complete documentation of testing and analysis performed to validate the operating system by injecting errors into the system during or prior to RROSS execution.

3.5.2) Availability

3.5.2.1) Data Protection

147	RROSS should provide user interfaces for storing data redundantly such that at least one copy is consistent if processing is stopped.
148	RROSS should provide user interfaces for storing data redundantly such that at least two copies are stored on physically distinct media, i.e. two different memory boards, a memory and a disk, etc.
149	RROSS should provide services which compare data stored redundantly and makes all copies equivalent to the latest consistent copy.
150	RROSS should provide services to report which task owns a specified allocated memory block.
151	RROSS should provide services to report which memory blocks are allocated to a specified task.
152	RROSS should provide a service to free all allocated memory blocks which are not marked to be available after the operating system restarts.

3.5.2.2) Restarting System

153	RROSS shall allow the user to handle a CPU reset when it is implemented in hardware as an interrupt.
154	RROSS shall provide the ability to restart the operating system and user programs asynchronously to their previous execution or completion.
155	RROSS shall provide the ability to restart the operating system without reloading the operating system image to memory.
156	RROSS shall not require the initialization of any memory except that containing critical data for the kernel before starting execution.
157	RROSS should provide services to reinitialize operating system data which is required to start the kernel from its initial state.

158	RROSS shall allow the user to create services to reinitialize operating system data which is required to start the kernel from its initial state.
159	RROSS shall allow the user to create services to reinitialize user data which is required to start the user programs from a saved state.
160	RROSS should provide services to allow user programs to invoke the restart of the operating system.
161	RROSS should provide services to allow user programs to invoke the restart of other user programs or themselves.
162	RROSS should provide services to allow user programs to invoke the restart of user programs and any child or sibling tasks.
163	RROSS shall allow the user to create services to store data which should be available though the operating system has been reinitialized and restarted between the storage and retrieval of data.
164	RROSS should provide services to store data which should be available though the operating system has been reinitialized and restarted between the storage and retrieval of data.
165	RROSS shall allow the user to provide services to find and retrieve data which was stored such that it would be available though the operating system has been reinitialized and restarted between the storage and retrieval of data.
166	RROSS should provide services to find and retrieve data which was stored such that it would be available though the operating system has been reinitialized and restarted between the storage and retrieval of data.
167	RROSS should provide services to checkpoint the state of a user program such that the program can be stopped and started from the saved state.

3.5.2.3) System Reconfiguration

168	The user should be able to specify the minimum needs of a system configuration which is considered valid and should include communication paths, reserved communication paths, number of processors, memory size, and reserve memory size.
169	If a valid system configuration exists (based upon specified needs), RROSS initialization services should find at least one valid configuration and provide it to the user programs.
170	If a valid system configuration exists (based upon specified needs), RROSS shall not preclude user initialization services from finding at least one valid configuration and providing it to the other user programs.

3.5.3) Security

3.5.3.1) Reporting Errors

171	RROSS services shall report detected errors to the caller of the service.

172	RROSS shall report the abnormal termination, or death, of a program due to an error, to the parent of the crashed program.
173	RROSS should provide services to query a program or resource for errors from another program.
174	RROSS services shall report Ada exceptions to the caller of the service.
175	RROSS shall report the death of a program due to an Ada exception, to the parent of the crashed program.
176	RROSS shall allow Ada exception handlers to use interprocess communications to report errors to other tasks.
177	RROSS shall allow Ada exception handlers to use interprocess communications to report errors to other tasks.

3.5.3.2) Detecting System Failures

178	RROSS shall allow the user to handle any and all hardware interrupts, including those reporting the detection of an error.
179	RROSS shall allow the user to maintain a *deadman* timer, i.e. a timer which requires resetting and, when expired, causes a CPU reset.
180	RROSS shall allow the user to maintain background health-and-status tasks which can access any memory by address and query the status of any executing task.

3.5.4) Maintainability

181	RROSS source code shall be available to users, subject to use restrictions (for reference, update, or both) as agreed between the RROSS supplier and user.
182	The RROSS supplier shall be responsible for long term support of the operating system version used by the spacecraft developer, subject to restrictions as agreed between the RROSS supplier and user.

3.5.5) Portability

183	RROSS shall be portable such that it can be implemented for a new processor architecture within one labor-year, including kernel, compilers, and all development tools.
184	RROSS should be modular in structure such that processor specific code is confined to 10% or less of the source code files used to build the operating system.
185	RROSS tools shall provide portability to users such that user programs can be used on any other architecture supported by RROSS by recompiling the user programs without change to the source code.
186	RROSS tools shall provide portability to users such that operating system interfaces are consistent between RROSS versions built for different architectures.

3.5.6) Configurability Guidelines

3.5.6.1) Using Optional Features

187	RROSS tools shall allow the user to build an operating system (RROSS) image which contains only those functions and data areas required by the user programs, i.e. RROSS shall be scaleable.
188	When removing a function from the RROSS image, the RROSS tools shall also remove any data and initialization routines which support only the removed function.

3.5.6.2) Creating Customized Features

189	RROSS tools should provide warning messages when system routines and functions are being replaced or redefined by a user written function.
190	RROSS tools shall allow the user to create functions which can be used as services available to the operating system and user programs.
191	RROSS tools shall allow the user to create functions which can be used to replace services provided by the operating system.

3.5.6.3) Specifying Configuration

192	RROSS tools should allow the user to specify the address at which programs and data are stored in the target computers memory.

DISTRIBUTION LIST

AUL/LSE 1 cy
Bldg. 1405 - 600 Chennault Circle
Maxwell AFB, AL 36112-6424

DTIC/OCP 2 cys
8725 John J. Kingman Rd Ste 944
FT Belvoir, VA 22060-6218

AFSAA/SAI 1 cy
1580 Air Force Pentagon
Washington, DC 20330-1580

AJPO 1 cy
Attn.: Joan McGarity
5600 Columbia Pike
Arlington, VA 22041

PL/SUL 2 cys
Kirtland AFB, NM 87117-5776

PL/HO 1 cy
Kirtland AFB, NM 87117-5776

Official Record Copy

PL/VTQ/ Capt Mary Boom 2 cys

Dr. R.V. Wick
PL/VT 1 cy
Kirtland AFB, NM 87117-5776

DEPARTMENT OF THE AIR FORCE
PHILLIPS LABORATORY (AFMC)

28 Jul 97

MEMORANDUM FOR DTIC/OCP
8725 John J. Kingman Rd, Suite 0944
Ft Belvoir, VA 22060-6218

FROM: Phillips Laboratory/CA
3550 Aberdeen Ave SE
Kirtland AFB, NM 87117-5776

SUBJECT: Public Releasable Abstracts

1. The following technical report **abstracts** have been cleared by Public Affairs for unlimited distribution:

PL-TR-96-1020	ADB208308	PL 97-0318 (clearance number)
PL-TR-95-1093	ADB206370	PL 97-0317
PL-TR-96-1182	ADB222940	PL 97-0394 and DTL-P-97-142
PL-TR-97-1014	ADB222178	PL 97-0300

2. Any questions should be referred to Jan Mosher at DSN 246-1328.

Jan Mosher

Jan Mosher
PL/CA

cc:
PL/TL/DTIC (M Putnam)